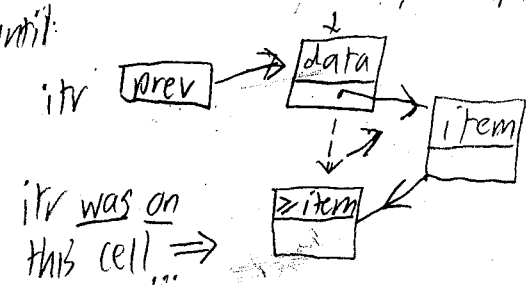
II Now consider an implementation with just the prev ptr.
The bodies of the ++ ops have `prev = prev->next;`
The linear-search code for a place to insert is unchanged until:



`itr.prev->next = new Cell(item, itr.prev->next);`
or with const - "under the hood"

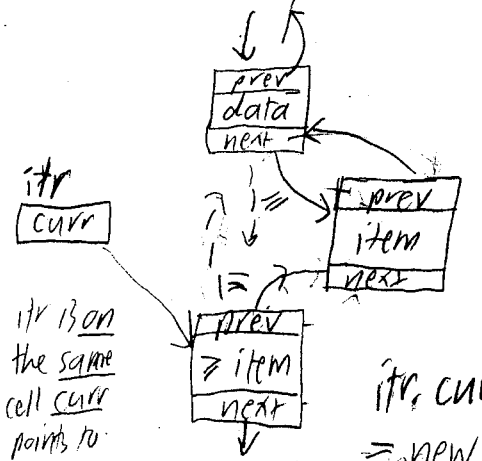
- Since there was just 1 field, no INV to worry about
- Since it's ok to change a field `prev->next` below top-level, this doesn't violate C++ "syntactic const"!

Hence we can keep "`const iterator& itr`".

- though since this version of insert is `private:`, it doesn't matter to the public!

However, "itr did change from being on the first data ≥ item (or end()) after the while loop, to being on the new item! And perhaps finishing with "return itr;" is not a totally safe idea. But one can do: `return iterator(itr.prev);`

III Doubly-Linked List Iterator – can use just a single curr pointer. To insert before:



Although "double assignments" are sometimes frowned upon, the one used to insert the new cell before itr is an idiom:

`itr.curr->prev = itr.curr->prev->next`
`= new Cell(itr.curr->prev, item, itr.curr);`

should in parameters, must in order of the : , , initializer syntax!

If your DNODE class orders fields with data before prev & next, your simple-setter constructor should/must use the same order.

Note: now this doesn't work if end is a NULL pointer, since then `itr.curr->prev` won't exist! Hence with a doubly linked list it is common to use a dummy end node. A dummy head node is not as important as with a singly-linked list, but without it you still have to treat insertion before begin() as a special case, since `itr.curr->prev->next` may not exist!

Either way, the itr stays where it was when the private insert was called (after the while loop), so you get both syntactic and "semantic" const! You need to return `iterator(itr.curr->prev)`, though. It seems itr does stay valid, but for erase it must not! (→ STL invalids demo)